# Nearest Neighbour Search with GPU

Ιακωβίδης Ιωάννης

26 Ιανουαρίου 2019

## 1 Introduction

The problem of nearest neighbour search in a space is of obvious importance in many fields. And the need for performance is increasing more and more. There are many algorithms for computing the search and designing data structure that improve efficiency. We will examine an implementation in three dimentions for GPU's using Cuda . As the technological growth in the industry has lead to amazing increase in performance through the use of GPU's.

## 2 Algorithm

For successful use the GPU we have to use an algorithm that can be greatly parallelized using the Cuda syntax (that is a direct result of the underlying hardware) and also can use memory efficiently. As the use of memory can dictate upper bounds in our performance. Our algorithm constructs a data structure from the set and uses it process quires. In detail we use a uniform grid of dimensions that relate to the size of the input. We assign the points of the set to the shells of the grid that they belong. And when we process the queries we start the search at the shell that they belong and branching out. If at any point we do not need to check other shells we break and return the current result.

## 3 Grid Construction

First we need to construct the grid from the set to help with our search. Let $C$ set be our original set, Q the set of queries and grid with dimensions $d \times d \times d$. We keep the coordinates of C and Q to there separate arrays $(C_x, C_y, C_z, Q_x, Q_y, Q_z)$ to have more coalesced memory accesses (Array of structures vs Structure of arrays rule). To put $C$ to the grid we really need to be able to reference for each shell the points that are in it. We can do this in place without additional space to store the points for each shell. We will sort C by grid shell keeping the start and end of each shell. Another advantage of this technique is that there a higher coalesced memory access rate, due to sorting.

### 3.1 Sorting

To sort by grid shell we need to generate a key for each shell. So we use a linear key assigning to shell $(i, j, k)$ the number $i * d * d + j * d + k$. This is being executed via the kernel:

```
__global__ void generatekey(float *Cx, float *Cy, float *Cz, int lenghtC,
  int *keys, int d)
```

After sorting by key is executed using the thrust library :

```
thrust ::stable_sort_by_key(kc, kc + lenghtC, make_zip_iterator(make_tuple
(Cx_ptr, Cy_ptr, Cz_ptr)));
```

The zip iterator is used so that we can group the three coordinates together and sort them with one function.

### 3.2 Finding Shell Boundries

To keep each shells boundaries we define two integer arrays starts and ends in the device. Such that starts[i] and ends[i] have the the start and end index of the points in the C arrays for the grid shell with key i. Starts and ends are both initialized to -1 so if we do not have points in a shell we will be able to identify it. We assign values to them via the kernel:

```
__global__ void findCellStartends(int *keys, int len, int *starts, int *ends)
```

Which is executed in parallel for all points in C. Looking at the previous point key if it is different than its key it starts a new shell and also looking at the next points key determines if it ends a shell.

## 4 Sorting Q and use of Shared Memory

After a number of runs we observe that sorting Q doesn't takes much time. And we can use the sorted Q to exposed more parallelism and use shared memory. So after we construct the grid. We generate keys for each point in Q and sort by those keys keeping the starts for each shell in the same way we did for C. Code for both implementations is included with and without this step.

## 5 Search

After the grid creation we can the search can be carried independently for each point independently for each point. For the code that doesn't use shared memory the search of the grid is implemented by the kernel :

```
__global__ void searchGrid(float *Qx, float *Qy, float *Qz, int *starts,
                           int *ends, float *Cx, float *Cy, float *Cz,
                           float *Resx, float *Resy, float *Resz, int d,
                           int lenghtQ)
```

It is launched so that every thread computes the search for a sequence of queries because sorting of queries doesn't happens in this implementation (the the reads of the point are not at all coalesced).

For each quire point the search starts at its cell finding the minimum and comparing to the minimum boarder distance. If the boarder distance is bigger than the minimum we store the result otherwise we continue our search to t neighbouring shells , such that we search the $3 \times 3 \times 3$ grid cube that includes our point. Then checking boarder distance we choose if we expand our search even more and so on.

The search that uses shared memory is implemented by the kernel:

```
__global__ void searchGrid(float *Qx, float *Qy, float *Qz, int *starts,
int *ends, float *Cx, float *Cy, float *Cz,
float *Resx, float *Resy, float *Resz, int d,int lenghtQ,int* startsQ )
```

That is launched so that every threadblock processes some number of grid shells. Meaning that it finds the result for each query in those grid shells. This is accomplished by building our point search specific code inside loop commands in the fashion that is shown in the next image:

```
for(int t=blockIdx.x;t<d*d*d;t=t+gridDim.x){
    if(startsQ[t]>=0){
    putShellinSharedMemory();
    __syncthreads();
    int  i=startsQ[t]+threadIdx.x;
    while(1){
        if(i>=lenghtQ){break;}
        int digit1 = Qx[i] * d;
        int digit2 = Qy[i] * d;
        int digit3 = Qz[i] * d;
        int key = d * d * digit1 + d * digit2 + digit3;
        if(key!=t){break;}
        searchOwnShell();//with help of shared memory
        searchOtherShells();//form global memory
        i=i+blockDim.x;
        }}}
```

In more detail each block gets a unique sequence of keys. Then if there exist points in Q for this shell we start the search process. First we make our shared memory (explained in detail below ) so we have as fewer loads as possible. Then giving to each thread a unique sequence of points until we go out of the array or the key changes (the point doesn't belong in the shell) we search.

The use of shared memory referenced above is done by putting every shell point in the shared memory arrays. So the search of the starting shell is done by searching this arrays in shared memory. If we need to search other shells we have to load for global memory. Attempts to load the whole $3 \times 3 \times 3$ cube have been tried but they didn't give any gain in performance. The use of shared memory is Dynamical so is the launch of the kernel we have to specify the size. But because the shells don't have necessarily the same number of points we needed to find the max. The is done in main by a reduction kernel:

```
__global__ void find_maximum_kernel(int *starts, int *ends, int *max,
                                    int *mutex, int n)
```

## 6 Validation and Performance

### 6.1 Validation

Validation is done by the functions:

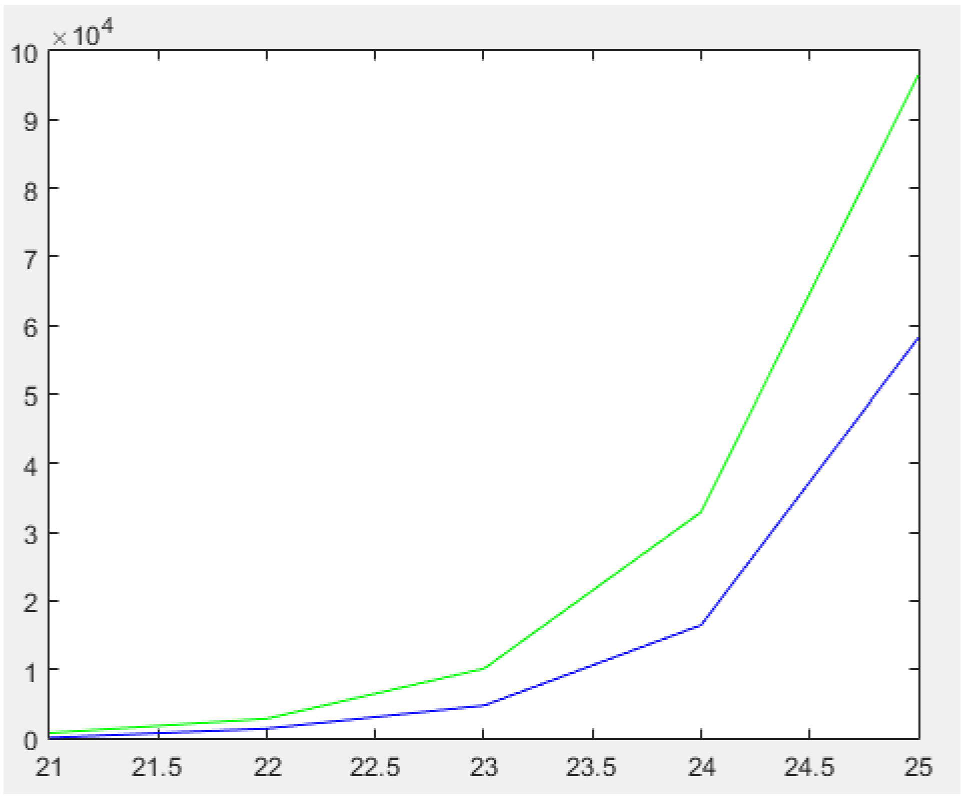
```
 __global__ void gridval(int *starts, int *ends, float *Cx, float *Cy, float *Cz,
                         int d)

__global__ void distsQmin(float *Qx, float *Qy, float *Qz, float *Cx, float *Cy,
                          float *Cz,int lenghtC, float *x, float *y, float *z,
                          int lenghtQ)
```

The first on validates the grid (sorting,starts and ends),while the second one the search results doing a brute force search. In the code that uses shared memory we validate the maximum search also with:

```
__global__ void checkmax(int max, int *starts, int *ends)
```

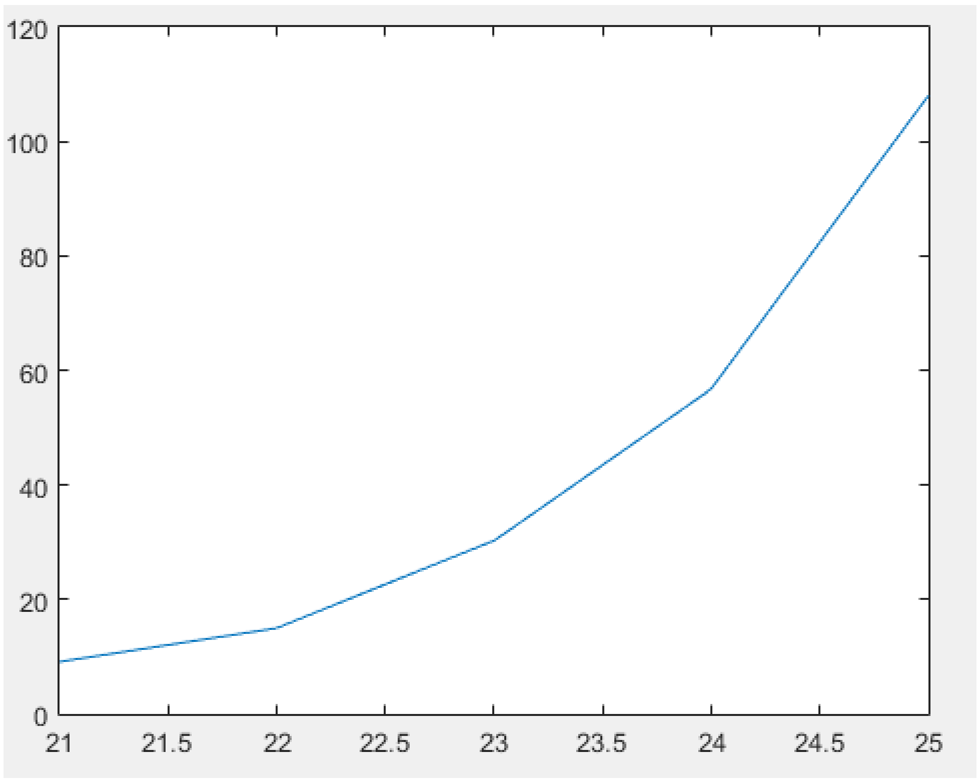
## 6.2 Performance

By timing the the execution we get the following results. First we see how the search is affected by the input size in the next graph the vertical axis is **time in milliseconds** and the horizontal is logarithm base 2 input size.

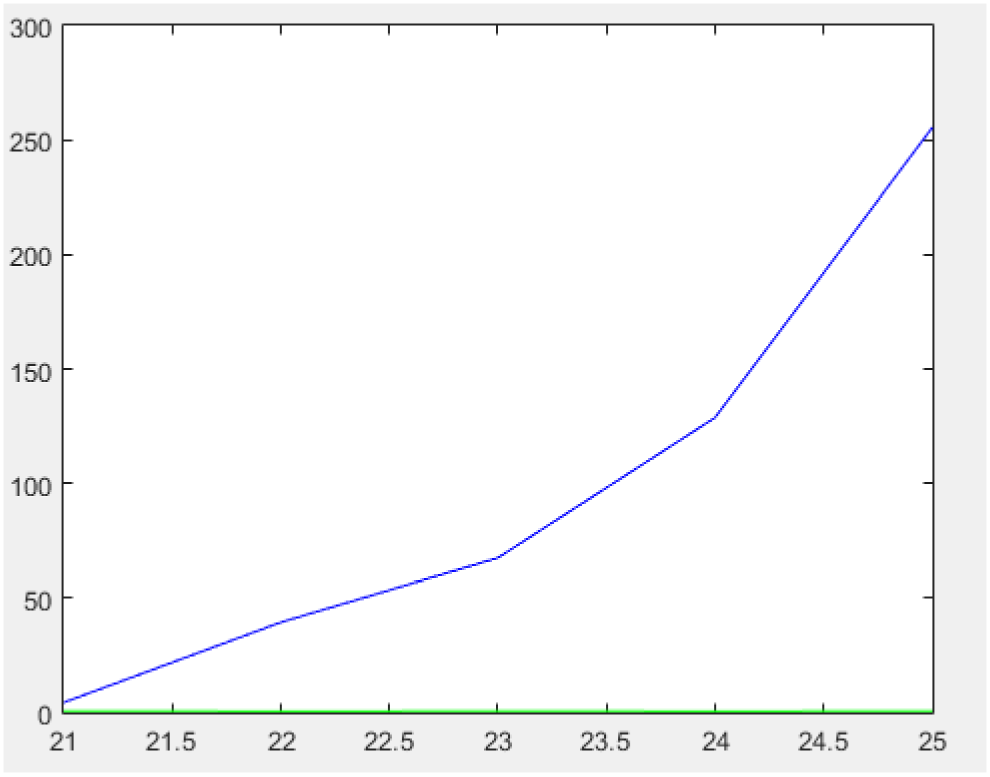


the green line is the search that uses shared memory the blue that doesn't so we see that we have a speed up of 2 in large inputs. In small inputs we have even greater speed up (for $2^{20}$ the non shared memory search is done in roughly 160 ms while the shared memory on in roughly 30 ms which give a speed up of more than 5 ). This is happens because the internal scheduler of cuda that schedules the blocks in the multiprocessors may not put above a certain number of blocks because we do not have that much memory.

As for the creation of the grid the graph bellow see how the input size affects the time for creating the grid (sorting plus finding the cell boundaries). Time is again in milliseconds.

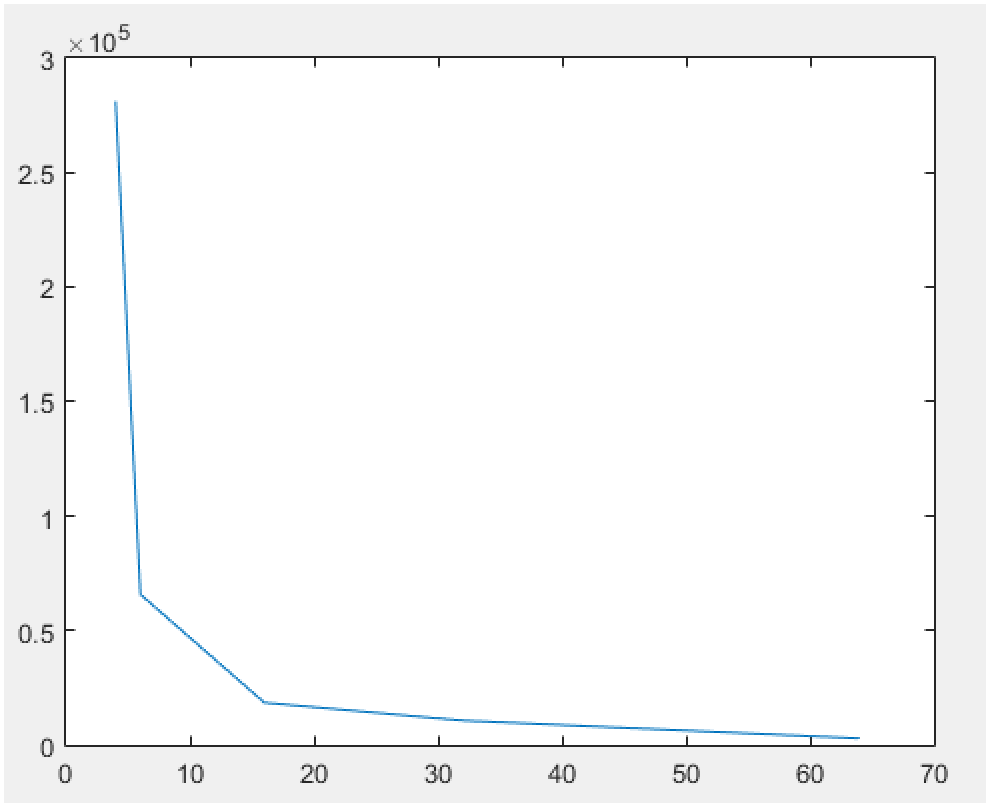


As we observe the time for the creation is significantly less than the time of the search. And comparing with the time of other techniques such that the validation process that finds every the minimum distance for all points serially in parallel. We can have a speed up to 300 (As 20759.328125 ms is a realistic validation time for $2^{20}$ ) Also for comparing the use of shared memory or not we have to take to account the additional operations that we do in order to make this use of shared memory (putting Q in the grid, sorting Q and find its cell boundaries ) And the following graph (sorting blue,find boundries green ) shows that are negligible compared to the searches.



Next we compare our algorithm with a previously implemented MPI Vantage Point tree search (link in bibliography ). And we find out that we obtain a speed up of 160 in the search operation and a 365 speedup over it using the shared memory implementation for input size of $2^{20}$.

Ending our performance analysis we see how the grid dimensions affect the grid construction and search. The grid construction remains totally unaffected by the change of the grid. While the search is getting faster as we see in the following graph for grid dimensions the horizontal axis (this is expected because we decrease the number of points that we search for each quire).



## 7 Code

Containing two files ”knn_shared.cu” shared memory implementation and ”knn.cu” no use of shared memory.
CodeLink

## 8 Bibliography

1. CUDA Toolkit Documentation
2. Trust Library Documentation
3. MPI-Vantagepoint Tree